Efficient Learning of Real Time One-Counter Automata

Efficient Learning of Real Time One-Counter Automata

Amr Fahmy
and
Robert Roos

TR-07-95

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Efficient Learning of Real Time One Counter Automata

Amr F. Fahmy*
Aiken Computation Lab.
Harvard University
amr@das.harvard.edu

Robert Roos
Department of Computer Science
Smith College
roos@sophia.smith.edu

April 14th 1995

Abstract

We present an efficient learning algorithm for languages accepted by deterministic real time one counter automata (ROCA). The learning algorithm works by first learning an initial segment, B_n, of the infinite state machine that accepts the unknown language and then decomposing it into a complete control structure and a partial counter. A new efficient ROCA decomposition algorithm, which will be presented in detail, allows this result. The decomposition algorithm works in $O(n^2 log(n))$ where nc is the number of states of B_n for some constant c.

If Angluin's algorithm for learning regular languages is used to learn B_n and the complexity of this step is $h(n, m)$ where m is the length of the longest counter example necessary for Angluin's algorithm, the complexity of our algorithm is thus $O(h(n, m) + n^2 log(n))$.

*Contact author. Address: Aiken Computation Lab., Harvard University, Cambridge, MA 02138 USA. Phone: 617 495-5841. Research supported in part by ARPA contract no. F19628-92-C-0113 and by NSF grant CDA-9308833.

1 Introduction

We present an efficient learning algorithm for languages accepted by deterministic real time one counter automata (ROCA). The learning algorithm works by first learning an initial segment, B_n, of the infinite state machine that accepts the unknown language and then decomposing it into a complete control structure and a partial counter. A new efficient ROCA decomposition algorithm, which will be presented in detail, allows this result. The decomposition algorithm works in $O(n^2 log(n))$ where nc is the number of states of B_n for some constant c.

If Angluin's algorithm for learning the regular languages is used to learn B_n and the complexity of this step is $h(n, m)$ where m is the length of the longest counter example necessary for Angluin's algorithm, the complexity of our algorithm is thus $O(h(n, m) + n^2 log(n))$.

Roos and Berman [BR87] and Roos [Roo88], were the first to find a polynomial time algorithm for the exact learning of Deterministic One Counter Automata (DOCA) as defined by Valiant in [VP75]. The polynomial is of large degree thus motivating this work to find a practical algorithm. Fahmy and Biermann [FB93] and Fahmy [Fah89] introduced the idea of learning by automata decomposition. The method is applicable to a very wide class of real time languages using a variety of data structures such as counters, stacks, queues and double counters, however the algorithms they present are of exponential time in the worst case. The definitions of control strcutures, data structures and behavior graphs that we use here and the relation between them were first given in [Bie77] and subequently in [FB93].

A discussion and an example of the learning process will be presented in Section 2. Following this, definitions of the control structure, the counter, the behavior graph and the relation between them are given in Section 3. In Section 4 the decomposition theorem and decomposition algorithm will be presented. Some notation and detailed proofs of lemmas are given in an Appendix.

2 The Learning Algorithm

We will present the learning process for a ROCA language using an example. Consider the language $L = \{a^n ccb^n d \mid n > 0\} \cup \{a^n d \mid n > 0\}$. This language is not regular and is accepted by an infinite state machine that we call the *behavior graph* (BG) and is denoted by B. An initial segment of B that includes all the states that are distance n or less from the initial state of B will be denoted by B_n. The BG for our example appears in Figure 1. Note that transitions to dead states are not shown.

A ROCA $A = (C, D)$ for L is a pair of state machines that also accepts L. C is a finite state machine called the *control structure* (CS) and D, called the *data structure*, is an infinite state machine that simulates the counter. State diagrams for a CS that accepts L and the counter also appear in Figure 1. L is accepted by A in the following manner. Input symbols are read by C which, using the symbol, its current state and the state of the counter, changes its states. While it is changing its state it sends a single instruction to the counter which it uses to change its state too. The triple (sym, val, instr) appear on the transitions of the CS in Figure 1 where sym is the input symbol, val is 0 if the counter state is 0 and $\neg 0$ otherwise. If the final symbol of an input string causes C to end up in a final state then we say that the ROCA has accepted the input string.

In this paper we show that B is *decomposable* into C and D. We give efficient algorithms to do this decomposition.

The learning process for a ROCA language L starts by constructing B_n, for some natural number n, for the unknown language. B_n is constructed using a slight modification of Angluin's algorithm for the regular languages using queries and counter examples. After constructing B_n and if n is large enough, the learning algorithm will be able to decompose B_n into a complete CS and a finite counter. The finite counter is then replaced with an infinite one and the learning algorithm will have constructed a complete ROCA for the unknown language. The value of n depends only on the language being learned; it could be small for very simple languages and large for complex ones.

We note that Angluin's algorithm will not terminate if the state machine that accepts L has an infinite number of states as in the case of ROCAs. Thus the teacher must choose a suitable depth n after which it asks the learner to decompose B_n. Also the teacher has to provide counter examples that will enable the learner to construct a complete B_n in the sense that no states will be missing from B_n.

After constructing B_n, the learner will attempt to decompose it into a complete CS and a finite counter.

The CS must be complete in the sense that no transitions from it are missing. The decomposition is done by performing what we call a *parallel breadth first search* (PBFS) from certain states. The PBFS marks the states of B_n, for our example, the PBFS starts at states $b1$, $b2$, $b3$, and $b4$.

The parallel searchs must mark isomorphic submachines of B_n. In our example these isomorphic graphs are labeled G_0, G_1, G_2 and G_3 in Figure 1. These markers are then used to construct partitions over the set of states of B_n that are necessary for the decomposition. Using the partitions it is then easy to construct the CS and the finite counter. To identify the states where the PBFS must start, a string w identifying a path from the initial state of B_n to an exit point in B_n must be identified. w is then broken into a string of the form xw' where w' is of the form y^k for some k. The PBFS starts from the states reachable using the strings xy^i $1 \leq i \leq k$. In our example there is only one exit point reachable using the string $w = a^5$ and w is broken into $x = a$ and $y = a$ and the PBFS starts at states $b1$, $b2$, $b3$, and $b4$. The BFSs are performed in a manner that guarantees that the resulting graphs are all isomorphic. In our example, the transitions $\delta(b1, c)$, $\delta(b2, c)$, $\delta(b3, c)$ and $\delta(b4, c)$ are all done in parallel. If for any one of the searches a transition is missing, then the PBFS is declared a failure. If the searches collide then we require that they all collide in a single state as in state bf for this example. The searches can run into each other only in a well defined manner where the search numbered i can collide with searches number $i + 1$ or $i - 1$. Again it is required that they all collide in the same way. In general, it is possible to have more than one exit point and the process would be repeated for each one.

In the reset of this paper we formalize the notions presented in this section and prove it correct.

3 Definitions and Properties of Real Time One Counter Automata

3.1 The Counter

Definition 3.1 A *counter* is a Moore type infinite state machine. The set of states of the counter is $N \cup E$ where N is the set of natural numbers and E is an error state. The initial state of the counter is state 0. The instructions, or input alphabet, of the counter is the set $\{i, d, n, r\}$ where i increments the state of the counter by one, d decrements the state by one, n is the no move or no change instruction and r is the reset instruction. Each state of the counter also has an output value associated with it. The initial state has output value 0, the error state has output value *error* and all the other states have output value $\neg 0$. The output function λ_D given a state, returns its output value. The state diagram of the counter appears in Figure 1. A formal definition of the counter is given in the Appendix.

A counter can be thought of as the memory of a computing device; it cannot, on its own, accept or reject input strings since it has no final states. It is a means by which a ROCA can count the occurrence of some event such as counting the number of times a certain string has appeared in the input for example.

3.2 The Control Structure

The second component of a ROCA is the *control structure* which is a deterministic finite state machine that reads input symbols from a finite non-empty alphabet Σ. Its next state depends on its own current state, the output value of the current state of the counter and the input symbol. The control structure issues instructions to the counter while it is changing states i.e. instructions to the counter appear on the control structure's transitions.

Definition 3.2 A *control structure* is a finite deterministic state machine $C = \{S_C, \Sigma_C, O_C, \delta_C, c_0, \lambda_C, F_C\}$ where:

- S_C is a finite nonempty set of states.
- Σ_C is a finite set of symbols called the *input alphabet* to C. Σ_C is the cross product of two alphabets; the finite nonempty *input alphabet* from the outside world, Σ, and the output alphabet of the counter O_D. Thus $\Sigma_C = \Sigma \times O_D$.
- O_C is a set of instructions that can be issued to the counter, $O_C = \Sigma_D = \{i, d, n, r\}$.

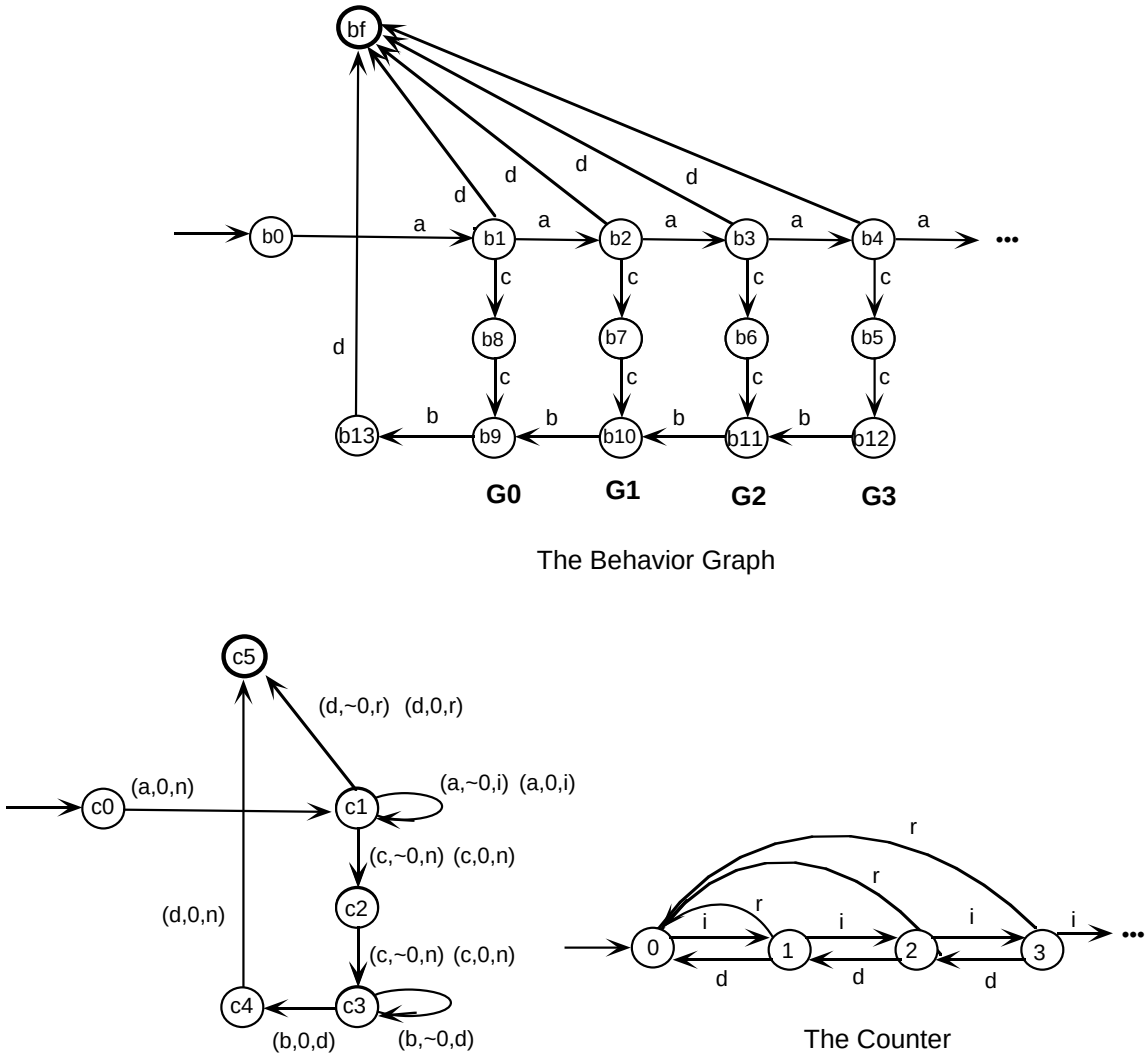


Figure 1: *The BG for the language* $L = \{a^n ccb^n d \mid n > 0\} \cup \{a^n d \mid n > 0\}$*, the state diagram for a CS for this language and the state diagram of the counter. Transitions to dead states in the BG and the CS are not shown. Also the error state of the counter and transitions from states to themselves are not shown for clarity.*

- $\delta_C : S_C \times \Sigma_C \longrightarrow S_C$ is a mapping by which C changes states called the *transition function* of C.
- c_0 is the *initial state* of C.
- $\lambda_C : S_C \times \Sigma_C \longrightarrow \Sigma_D$ is a mapping that assigns to each transition of C an instruction to the counter called the *instruction assignment function*.
- $F_C \subset S_C$ is a set of *final states* of C.

3.3 Real Time One Counter Automata

A *Real Time One Counter Automaton* (ROCA) A is a pair of state machines (C, D) where C, is a control structure and D, the data structure, is an infinite state machine that is isomorphic to a counter. A ROCA reads one input symbol at a time. After reading each symbol the CS changes states from its current state by examining the output symbol of the current state of the counter and using its transition function it makes the transition to the next state. While it is doing so, it sends exactly one instruction from the set $\{i, d, n, r\}$ to the counter. The counter receives the instruction and executes it. The ROCA is now ready for a new symbol. If the input symbol has caused the CS to be in a final state then we shall say that the ROCA has accepted the input string, otherwise we say that the input string has been rejected.

The following definitions explain, in a more formal manner, how a ROCA works and how it accepts strings. An *instantaneous description* (ID) of a ROCA is given by the pair $< c, d >$ where c and d are the current states of its CS and DS respectively. The *initial ID* of the ROCA is the ID $< c_0, d_0 >$. If the current ID of ROCA A is $< c, d >$, the following events take place when A reads an input symbol α:

$$\begin{array}{ll} \delta_C(c, (\alpha, \lambda_D(d))) = c' & \text{(Compute next state of C)} \\ \lambda_C(c, (\alpha, \lambda_D(d))) = I & \text{(Send instruction to D)} \\ \delta_D(d, I) = d' & \text{(Compute next state of D)} \end{array}$$

A ROCA A *accepts* a string $\sigma \in \Sigma^*$ iff the sequence of transitions induced by the letters of σ leads from $< c_0, d_0 >$ to $< c', d' >$ where $c' \in F_C$ and the counter never enters its error state. Let $\mathcal{L}(A)$ denote the language accepted by A, that is $\mathcal{L}(A) = \{\sigma \in \Sigma^* \mid A \text{ accepts } \sigma.\}$ Such a language will be called a ROCA language.

3.3.1 The Real Time Constraint

From the previous definitions we can see that the CS of a ROCA, when given an input string of length n, can issue exactly n instructions, some of which could be the null or do nothing instruction, to the counter; one instruction per input symbol. So, for example, if the current state of the counter is d and if the current input string is of length n, then after the CS reads the input string, it can leave the counter in a state d' such that

$$d - n \leq d' \leq d + n$$

In this respect the ROCA is more restricted than the DOCA since DOCAs can change the value of the counter by more than one per input symbol and also because ROCAs are not allowed to make epsilon transitions.

3.4 The Behavior Graph of a ROCA

A ROCA works in much the same way as finite state machines that accept the regular languages work. It consumes one input symbol at a time and after each such symbol we can determine if it has accepted the input. We shall now define a single infinite state machine that can do the work that a ROCA does.

Definition 3.3 The *behavior graph* (BG) of a ROCA A is the reduced Moore type state machine

$$B = \{S_B, \Sigma, \delta_B, b_0, F_B\}$$

that accepts $L = \mathcal{L}(A)$.

The BG is reduced in the sense that among its states, no two are equivalent. The BG of a ROCA is unique, up to isomorphism. It is a state machine with infinite number of states.

If π_L is the right invariant equivalence relation defined by L over Σ^*, $\beta_{\pi_L}(\sigma)$ is the block of π_L that contains σ and $T_L(\sigma)$ is the set of suffixes of σ with respect to L, then B is defined as follows:

- $S_B = \pi_L$, the set of states of the BG is the set of blocks of the partition π_L where $\sigma_1 \equiv \sigma_2\ (\pi_L)$ iff $T_L(\sigma_1) = T_L(\sigma_2)$.
- Σ is the same input alphabet as that of A.
- $\delta_B : S_B \times \Sigma \longrightarrow S_B$ and is defined as follows $\delta_B(\beta_{\pi_L}(\sigma), \alpha) = \beta_{\pi_L}(\sigma\alpha)$
- $b_0 = \beta_{\pi_L}(e)$, where e is the null string. The initial state of B is the unique block of π_L that contains the null string.
- $F_B = \{\beta_{\pi_L}(\sigma) \mid e \in T_L(\sigma)\}$. The set of final states of B is the set of blocks of π_L such that the null string is a suffix for strings in the block.

δ_B is well defined since π_L is a right invariant equivalence relation i.e. if $\sigma_1 \equiv \sigma_2\ (\pi_L)$ then $\forall\alpha \in \Sigma, \sigma_1\alpha \equiv \sigma_2\alpha\ (\pi_L)$. A string σ is accepted by a BG if it drives B from its initial state b_0 to a state $b \in F_B$, i.e. if the null string is in $T_L(\sigma)$.

So far we have two different views of the language accepted by a ROCA; the CS and the counter of the ROCA working together to accept strings in the language and the BG that accepts the language. In the next section we will relate the two views. Our goal is to show that the CS and DS are a *decomposition* of the BG.

3.5 The Product of a Control Structure by the Counter

Let $A = \{C, D\}$ be a ROCA, the product of the CS by the counter is going to be a third machine $M_{C\times D}$. The set of states of the product machine is a subset of $S_C \times S_D$. The alphabet of the product machine is going to be the input alphabet, Σ of the ROCA. To connect state (c, d) to state (c', d'), there must be a transition from c to c' in C and a transition from d to d' in D. The instruction labeling the transition from c to c' must be the same as that leading from d to d'. Also the transition from c to c' must be labeled with a test value from O_D which is the same as that of d. The initial state of the product machine is (c_0, d_0). State (c, d) will be labeled as a final state of the product machine if c is a final state of C. Notice that the product machine will contain many useless states, i.e. states that are unreachable and states that have no tails. If all the useless states are deleted from the set of states of the product machine, it will not affect the language it accepts. Formally,

Definition 3.4 For a ROCA, $A = \{C, D\}$ where

$$C = \{S_C, \Sigma_C, O_C, \delta_C, c_0, \lambda_C, F_C\} \text{ and } D = \{S_D, \Sigma_D, O_D, \delta_D, d_0, \lambda_D\}$$

the *product* $C \times D$ is the Moore type state machine $M_{C\times D} = \{S_{C\times D}, \Sigma, \delta_{C\times D}, p_0 F_{C\times D}\}$ where

- $S_{C\times D} \subseteq S_C \times S_D$.
- $\forall\alpha \in \Sigma\ \delta_{C\times D}((c, d), \alpha) = (\delta_C(c, (\alpha, \lambda_D(d))), \delta_D(d, I))$ where $I = \lambda_C(c, (\alpha, \lambda_D(d)))$.
- $p_0 = (c_0, d_0)$.
- $(c, d) \in F_{C\times D}$ if $c \in F_C$.

The following theorem characterizes the language accepted by $M_{C\times D}$ and relates it to B.

Theorem 3.1 Let $A = \{C, D\}$ be a RTA with BG B, we have

$$\mathcal{L}(B) = \mathcal{L}(M_{C\times D}).$$

Proof: A proof is immediate by induction on the length of strings accepted by A and $M_{C\times D}$. □

Since the BG of a ROCA is equivalent to the product machine, there is an equivalence between the states of the two machines. Recall that two states are equivalent if the two states have the same set of tails. Every useful state, (c,d), of $M_{C\times D}$ is equivalent to some state b of the BG and every state, b of the BG is equivalent to some useful state (c,d) of $M_{C\times D}$. If state b is equivalent to state (c,d) we shall write $b=(c,d)$. It is going to be very useful to think of every state of the BG as having two components; a CS-component and a DS-component.

3.6 The Repetitive Structure of the Behavior Graph

Define the mapping g from the set of configurations to the set of states in the BG so that each state of $M_{C\times D}$ is mapped to the state corresponding to it. Then $g(p,i)=g(q,j)$ iff (p,i) is equivalent to (q,j). In other words, g maps configurations, (c,d)-pairs into their equivalence classes, which, in turn, are the states of the BG. Extend g to submachines in the obvious way, i.e., map transitions using $g(\delta_{C\times D}((p,i),a))=\delta_B(g(p,i),a)$. Then a subgraph of $M_{C\times D}$ is mapped into a subgraph of the BG.

Let X be a set of configurations, let $G(X)$ be the graph induced by X, i.e., the portion of the configuration graph containing X and all transitions among elements of X. Let $X_{m,n}$ be the set of reachable configurations $\mathrm{n}X_{m,n}=(p,i):p\in S_C, m\le i<n$ We call this a "slice" of the configuration graph; it consists of all configurations with counter values bounded above and below by fixed constants.

The overall goal is to prove that:

> There exist two constants, H and K, such that the graphs
>
> $$G(X_{H,H+K}), G(X_{H+K,H+2K}), G(X_{H+2K,H+3K}), \ldots.$$
>
> all have isomorphic images under the mapping g. Moreover, either all these images coincide, or they are all distinct subgraphs of the BG.

Because a ROCA can distinguish only between zero and nonzero counter values, the infinite transition diagram of the automaton $M_{C\times D}$ has a special kind of translational invariance with respect to the counter value. More precisely, if $\delta_{C\times D}((p,i),a)=(q,j)$, for $i,j>0$, then $\delta_{C\times D}((p,i+k),a)=(q,j+k)$ for any $k>0$, and if $\delta_{C\times D}((p,i),a)=(q,0)$ for some $i>1$, then $\delta_{C\times D}((p,i+k),a)=(q,0)$ for all $k>0$.

We now show that we can assign artificial counter values to the states of the behavior graph B to give it a similar repetitive structure; one way to do this is by examining the equivalence classes of the reachable states in $M_{C\times D}$. We will drop the subscript "$C\times D$" on our transition function δ in the following discussion.

We need the following fact, whose proof should be self-evident:

> If $\delta((p,i),w)$ is final and $\delta((p,j),w)$ is nonfinal, for some $i,j>0$, then at least one of the two computations on w must visit a state $(q,0)$ without performing a reset.

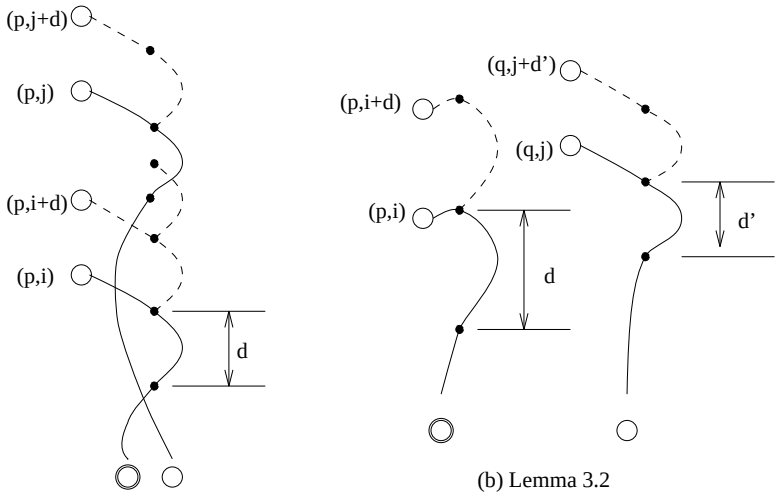
The next fact is an immediate consequence of the preceding one:

> For each $w\in\Sigma^*$, for each $p\in S_C$, there is a threshold value t (no larger than $1+|w|$) such that the states $\delta((p,t+i),w)$ are all final or are all nonfinal for every $i\ge 0$.

Lemma 3.1 For each $p\in S_C$, there are constants t and k such that for each j, $0\le j<k$, either:

- all states $(p,t+j+rk), r\ge 0$, are pairwise nonequivalent, or
- all states $(p,t+j+rk), r\ge 0$, are equivalent

> **Proof sketch.** A detailed proof appears in the Appendix.
>
> Suppose there exist $i,j>|S_C|$ and string w such that $\delta((p,i),w)$ and $\delta((p,j),w)$ consist of a final and a nonfinal state. By "pumping" a certain substring of w we can obtain pairwise nonequivalence for an infinite set of states $\{(p,r_i)\}$ where the r_i belong to an infinite index set with a regular structure parameterized by a constant $d\le|S_C|$. Iterate this process. We achieve the lemma with threshold $t=\max\{\text{lengths of all witnesses used}\}$ and $k=$ least common multiple of all the ds. (See figure 2(a).) □



(a) Lemma 3.1

(b) Lemma 3.2

Figure 2:

Corollary 3.1 There is some threshold t and constant k such that, for any $p \in S_C$, the results of the previous lemma hold.

Proof. Repeat the lemma for each state; take the largest threshold and the least common multiple of all the ks. □

Lemma 3.2 Let p and q be any two states. There is a threshold t and a constant k such that, for each $i, j, 0 \leq i, j < k$, either:

- $(p, t + i + mk)$ and $(q, t + j + mk)$ are nonequivalent for each $m \geq 0$; or
- $(p, t + i + mk)$ and $(q, t + j + mk)$ are equivalent for all $m \geq 0$.

Proof sketch. A detailed proof appears in the Appendix. Suppose we have a witness w to the nonequivalence of some state pair (p, i) and (q, j). In the only hard case that needs to be considered, we consider the two witness computations side-by-side. Some pair of states must repeat, forming "descending loops" in both computations over the same substring of w. By a case analysis, we can show that the family of witnesses obtained by "pumping" the loop will yield infinitely many nonequivalences of p- and q-state pairs above a certain threshold t. (See figure 2(b)). □

We can now observe the desired regularity condition in B by assigning the labels to its states. Let t and k be the constants guaranteed by the lemma. For each state s of B, consider the set $E(s)$ of states of $M_{C \times D}$ that are equivalent to s. Uniformly choose some representative state (p, i) from $E(s)$ (e.g., the lowest-numbered state in the equivalence class, and the lowest counter value appearing with that state). If $E(s)$ is an infinite set, or if it contains only states whose counter values are less than the threshold t determined by the last lemma, assign a unique state name and a counter value of zero to s. Otherwise, i is of the form $t + kr + j$, so assign s the name $[p, j]$ and the counter value r. The translational invariance now holds.

Now that we know that a labeling exists which establishes the regular structure of B, we know that any search for such a labeling will take place in a nonempty search space (although we will obtain our labels in a different way).

4 Decomposing the Behavior Graph of a ROCA

Hartmanis and Stearns [HS66] developed a theory for the decomposition of finite state machines in the early sixties. By decomposing a state machine into a number of state machines that emulate the behavior of the

composite machine they were able to use less electronic components to realize the machine. They invented what is known as partition pairs, see the Appendix for a definition, The basic idea behind decomposition is to merge states, and thus the use of partitions, while preserving the transitions. Merged states in one machine must not be merged in at least one other component machine. In this section we discuss the decomposition theorem that will tell us exactly when a decomposition is possible.

4.1 The Associated Partitions of a ROCA

Given a ROCA and its BG, Theorem 3.1 states that every state of the BG is equivalent to a state of $M_{C\times D}$ and vice versa. We shall define two partitions over the states of BG that group states with the same components in the same block. One partition groups states of the BG that have the same CS component in the same block and the second groups states of the BG that have the same counter component in the same block. Using these two partitions we can decompose the BG into a CS and a counter.

Definition 4.1 Given a ROCA $A = \{C, D\}$ with BG B, let b_1 and b_2 be two states of the BG, c_1, c_2 be states of the CS and d_1, d_2 be two states of the counter such that

$$b_1 = (c_1, d_1) \quad \text{and} \quad b_2 = (c_2, d_2)$$

We define the four partitions π_C, π_D, ρ_{CD} and ρ_{DC} over the set of states of B as follows

- $b_1 \equiv b_2\ (\pi_C)$ iff $c_1 = c_2$.
- $b_1 \equiv b_2\ (\pi_D)$ iff $d_1 = d_2$.
- $b_1 \equiv b_2\ (\rho_{CD})$ iff $\forall \alpha \in \Sigma, \forall v \in O_D, \lambda_C(c_1, (\alpha, v)) = \lambda_C(c_2, (\alpha, v))$. Two states of B share the same block of ρ_{CD} iff, for every letter of the alphabet and every test value of the DS, the CS-components of the behavior states send the same instructions to the DS.
- $b_1 \equiv b_2\ (\rho_{DC})$ iff $\lambda_D(d_1) = \lambda_D(d_2)$. Two states of B share the same block of ρ_{DC} iff the DS-components of the two behavior states have the same test value.

The four partitions are called the *associated partitions* of ROCA A over the set of states of B. They tell us which states of the BG will have the same properties when B is decomposed into C and D.

4.2 The Decomposition Theorem

The following theorem tells us exactly when a BG can be decomposed into a CS and a counter given a set of partitions over the set of states of the BG. This theorem is a special case of a theorem that appeared in [FB93], [Fah89] and [HS66] and thus will be stated here without proof. Also the reader is refered to the Appendix for definitions and properties of partitions and partitions pairs.

Theorem 4.1 Let A be a ROCA with BG B. Let π_C, π_D, ρ_{CD} and ρ_{DC} be four partitions over the set of states of B. The four partitions are the associated partitions of ROCA A over the BG B iff

(i) . $(\pi_C.\rho_{DC}, \pi_C)$ is a partition pair.

(ii) . $(\pi_D.\rho_{CD}, \pi_D)$ is a partition pair.

(iii) . $\pi_C \leq \rho_{CD}$

(iv) . $\pi_D \leq \rho_{DC}$

(v) . $\pi_C.\pi_D = \mathbf{0}$

(vi) . The number of blocks of π_C is finite.

(vii) . $\pi_C \leq \pi_F$, i.e. π_C is output consistent.

(viii) . The machine $D = \{S_D, \Sigma_D, O_D, \delta_D, d_0, \lambda_D\}$ is isomorphic to a counter where D is defined by

- $S_D = \pi_D$.
- $\Sigma_D = \rho_{CD} \times \rho_{DC} \times \Sigma$.
- $O_D = \rho_{DC}$.
- $d_0 = \beta_{\pi_D}(b_0)$.
- $\lambda_D(\beta_{\pi_D}) = \beta_{\rho_{DC}}(\beta_{\pi_D})$.
- $\delta_D(\beta_{\pi_D}(b), (\beta_{\rho_{CD}}(b), \beta_{\rho_{DC}}(b), \alpha)) = \beta_{\pi_D}(\delta_B(b, \alpha))$

4.3 A Decomposition Algorithm

Given B_n, our aim now is to describe an algorithm that will mark states of B_n with (c, d)-pairs such that the necessary partitions for the decomposition can be created.

Since B_n is a submachine of B there are transitions in B_n that lead to states that are not in B_n, such transitions will be called *exit points*.

Let w be a shortest string that leads from the initial state of B_n to an exit point e in B_n. Such strings can be found in an efficient manner. The following algorithm uses w to identify a periodic structure in B_n (and, hence, B).

for each prefix x of w **do**
    let $w = xw'$
    let $p_0 = \delta_{B_n}(q_0, x)$
    for each prefix y of w' **do**
        let $p_i = \delta_{B_n}(p_0, y^i)$ for $i = 1, 2, \ldots$, until
        $p_i = e$ for some i or until a state is repeated.
        if e reached in previous step **then**
            PARBFS$(p_0, p_1, \ldots)$
            if success **then** exit;
            end if
        end if
    end for
end for

The outer loop searches for a prefix of w sufficiently long to reach states above some (unknown) threshold t. The inner loop searches for a pair of states, p_0 and p_1, that are isomorphic under some periodic description of B, then uses the substring y to try to determine additional isomorphic copies of these states. Procedure **PARBFS** is described below; it performs breadth-first searches "in parallel" from each of the states $p_1, p_2, \ldots$ and succeeds if it detects a periodic structure. (The precise mechanism for simulating this parallelism is unimportant.) Searches are synchronized so that, for any string m that labels a path from the root of a BFS search tree, the searches simultaneously visit the nodes $\delta_{B_n}(p_i, m)$, for $i = 0, 1, \ldots$.

A single breadth-first search from state p_i marks nodes with a "search number" i and a word m (which describes the path used to reach that node). A BFS halts when it can't extend the search without using nodes that are already labelled (possibly by one of the searches operating in parallel with it). Each individual BFS is a standard queue-based breadth first search; the search queue contains (state,string) pairs, and initially contains (p_i, ϵ). When we remove (q, m) from the queue, we label it (i, m) and then, for each $a \in \Sigma$ and each neighbor r of q, we add (r, ma) to the queue if r has not already been labelled.

Two graphs G_i and G_j visited by BFS(p_i, i) and BFS(p_j, j) are called isomorphic if there is a one-to-one correspondence f between their nodes that preserves membership in F_n, the set of final states, and such that $p = f(q)$ if and only if, for some constant c and for each $a \in \Sigma$, $\delta_{B_n}(p, a) = f(\delta_{B_n}(f(q), a))$, the label of p is (i, m) and the label of q is $(i + c, m)$ (for some string m). A finite sequence of graphs $G_0, G_1, \ldots$ satisfies the property **isom** if there is a nonempty subsequence $G_i, G_{i+1}, G_{i+2}, \ldots$ such that G_i is isomorphic to G_{i+1} with constant $c = 1$.

Finally, here is **PARBFS**; k is a small constant which may be used to ignore "bad" searches caused by being too near the zero counter states or too near the exit points:

PARBFS$(p_0, p_1, \ldots)$**:**
for $i = 0, 1, \ldots$ **paralleldo**
    $G_i =$ **BFS**(p_i, i);
    if $k < i < n - k$ and G_i and G_{i+1} violate the **isom** property **then**
        return failure
    end if
end for
return success;

4.4 Complexity OF The Learning Algorithm

We measure the complexity of the learning algorithm in terms of the number of states of B_n, nc for some constant c. Assume that Angluin's algorithm is used to construct B_n and assume that the complexity of this step is in $O(h(n,m))$ steps where m is the length of the longest counter example necessary for this algorithm. The decomposition step first searches for a string w to an exit point which can be found in $O(n)$ steps. The string w is then partitioned into the form xw' which can be done in $O(n)$ steps. For each such x, w' is partitioned into a string of the form y^i in another $O(n)$ steps for a total of $O(n^2)$ steps to rewrite w in the form xy^i. See [ML84] for an $O(nlog(n))$ to perform this last step. For each such form of w, a PBFS must be performed. If there are k such BFSs then each must be of depth n/k and the complexity is $O(n)$. The complexity of the decomposition step is thus $O(n^3)$ or $O(n^2log(n))$ if the algorithm in [ML84] is used. The complexity of the learning algorithm is thus $O(h(n,m) + n^2log(n))$.

References

[Bie77] A.W. Biermann. A fundamental theorem for real time programs. Technical report, Duke Univerity Department of Computer Science, 1977.

[BR87] P. Berman and R. Roos. Learning one-counter languages in polynomial time. *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 61–67, 1987.

[Fah89] A. F. Fahmy. *Synthesis of Real Time Programs.* PhD thesis, Duke University, 1989.

[FB93] A. F. Fahmy and A. W. Biermann. Synthesis of real time acceptors. *Journal of Symbolic Computation*, 15:807–842, 1993.

[HS66] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, 1966.

[ML84] M. Main and R. Lorentz. An O(nlog(n)) algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–432, 1984.

[Roo88] R. Roos. *Deciding Equivalence of Deterministic One-Counter Automata in polynomial time with applications to learning.* PhD thesis, The Pennsylvania State University, 1988.

[VP75] L. G. Valiant and M. S. Paterson. Deterministic one-counter automata. *Journal of Computer and System Sciences*, 10:340–350, 1975.

Appendix

Formal Definition of the Counter

Formally, the counter is defined by

$$D = \{S_D, \Sigma_D, O_D, \delta_D, d_0, \lambda_D\}$$

where

- $S_D = N \cup E$, where N is the set of natural numbers and E is an error state.
- $\Sigma_D = \{i, d, n, r\}$.
- $O_D = \{0, \neg 0, error\}$.
- δ_D is defined by: $\forall x \in N$,

$$\begin{aligned}
\delta_D(x, i) &= x + 1 \\
\delta_D(x, n) &= x \\
\delta_D(0, d) &= E \\
\delta_D(x, d) &= x - 1, \quad x \neq 0 \\
\delta_D(x, r) &= 0 \\
\delta_D(E, I) &= E \ \ \forall I \in \{i, d, n, r\}
\end{aligned}$$

- $d_0 = 0$.
- $\lambda_D(d)$ is the output function that assigns an output value from O_D to each state d and is given by: $\forall x \in N, x \neq 0$

$$\begin{aligned}
\lambda_D(x) &= \neg 0 \\
\lambda_D(0) &= 0 \\
\lambda_D(E) &= error
\end{aligned}$$

Partitions and Partition Pairs

A *partition*, π, on a set S is a collection of pairwise disjoint subsets of S whose union is S. Each subset is called a *block* of the partition. If two elements, s and t of S, are in the same block of π, we shall write $s \equiv t \ (\pi)$. The block of π containing an element s will be denoted by $\beta_\pi(s)$. If π_1 and π_2 are two partitions on a set S then the *product* of π_1 and π_2 denoted by $\pi_1.\pi_2$ is also a partition on the set S such that $s \equiv t \ (\pi_1.\pi_2)$ iff $s \equiv t \ (\pi_1)$ and $s \equiv t \ (\pi_2)$. The partition that puts every element in a block by itself is called the *zero partition* and will be denoted by $\mathbf{0}$. If π_1 and π_2 are two partitions on a set S, then π_2 is said to be *larger than or equal to* π_1, denoted by $\pi_1 \leq \pi_2$, if every block of π_1 is a subset of a block of π_2. It is clear that $\pi_1 \leq \pi_2$ iff $\pi_1.\pi_2 = \pi_1$.

Let π be a partition over the set of states of some machine M. π will be called *output consistent* iff $b \equiv b'(\pi)$ implies that either both states are final or both are non final. The largest output consistent partition will be denoted π_F and is the unique partition that groups all the final states in one block and all non-final states in another block.

Definition 4.2 Let π and π' be two partitions on the set of states, S, of some machine M. The ordered pair (π, π') is called a *partition pair* (pp) iff

$$\forall s, t \in S, \forall \alpha \in \Sigma, s \equiv t \ (\pi) \Rightarrow \delta(s, \alpha) \equiv \delta(t, \alpha) \ (\pi')$$

If (π, π') is a pp then from the block of π that contains the current state of the machine we can find the block of π' that contains the next state on every letter of the alphabet. We can see that $\delta(\beta_\pi, \alpha) \subseteq \beta_{\pi'}$.

A partial ordering, $\leq$, is defined on pps by comparing the respective components of pps. If (π, π') and (τ, τ') are two pps then $(\pi, \pi') \leq (\tau, \tau')$ iff $\pi \leq \tau$ and $\pi' \leq \tau'$.

See [HS66] for more properties of partition pairs.

Proof of Lemma 3.1

Suppose there exist $i_1, j_1 > |S_c|$ such that (p, i_1) and (p, j_1) are nonequivalent. Let w_1 be a (smallest) witness to their nonequivalence. By the pigeonhole principle applied to a sequence of states with decreasing counter values, the computation paths traced by $\delta((p, i_1), w_1)$ and $\delta((p, j_1), w_1)$ must contain a *descending loop* (a path that loops in the control structure while decreasing the counter). Write $w_1 = \alpha_1\beta_1\gamma_1$, where $\beta_1 \neq \epsilon$ is the loop. The "drop" d_1 of β_1 can be bounded by $|S_C|$ (consider only the states in the computation path that have counter components between $i_1 - |S_C|$ and i_1). Consider the infinite family of witnesses $\{\alpha_1\beta_1^i\gamma_1 : i \geq 0\}$. These can be used to prove that $(p, i_1 + t_1 + rd_1$ is not equivalent to $(p, j_1 + t_1 + sd_1)$ for any $r, s \geq 0$, for some threshold t_1 bounded by $|w_1| + 1$. In addition, for at least one of the two values i_1 and j_1, say i_1, we have the pairwise nonequivalence of all states in the set $\{(p, i_1 + t_1 + rd_1) : r \geq 0\}$. (The latter follows from the fact that, because w_1 was chosen smallest, $\alpha_1\gamma_1$ is not a witness between (p, i_1) and (p, j_1); hence, it *is* a witness between either (p, i_1) and $(p, i_1 - d_1)$ or (p, j_1) and $(p, j_1 - d_1)$; now we can "pump" β_1.)

If an infinite number of nonequivalences among p-states remain unaccounted for, repeat this construction on other pairs of p-states (p, i_r), (p, j_r), $r = 2, 3, \ldots$, obtaining witnesses w_r. This process cannot continue forever; at any stage, the set of p-states containing nonequivalent pairs that have not been accounted for will be a union of periodic (with respect to the counter) sets, with period at most the least common multiple of all of the loop drops used so far. The above construction shows that some infinite periodic subset of this, with period bounded by $|S_C|$, will be covered in the next step. Therefore, we will achieve the lemma with a threshold t that is no more than the maximum of all $|w_r| + 1$ and a value of k that is no worse than the least common multiple of the d_r.□

Proof of Lemma 3.2

Choose constants t_0 and k_0 using the construction of corollary 3.1. Suppose neither condition of the lemma holds. Then we have a witness w_1 to the nonequivalence of some state pair (p_1, i_1) and (q_1, j_1), where $i_1, j_1 > t_0$. Consider the sequence of state pairs visited during the computations of $\delta((p_1, i_1), w_1)$ and $\delta((q_1, j_1), w_1)$. If neither contains a state with zero counter value, or if both reach zero states using the reset operation, we can extend the nonequivalence to infinitely many state pairs $(p_1, i_1 + r)$, $(q_1, j_1 + s)$ using the same witness. Therefore, assume at least one computation descends to zero by counting down. If t_0 is at least $|S_C|^2$, some pair of states repeats in the two computations, forming loops in both computations over the same substring β_1. This repetition can be chosen so that the loop is descending in at least one of the computations (specifically, the one that "counts down").

Let $w_1 = \alpha_1\beta_1\gamma_1$. For each state $r \geq 0$, let m_r denote the smallest value of m such that $\{\delta((p_1, i_1 + d_1r), \alpha_1\beta_1^m\gamma_1), \delta((p_1, i_1 + d_1r), \alpha_1\beta_1^{m-1}\gamma_1)\}$ contains both a final and a nonfinal state. Let n_r be the smallest n such that $\{\delta((q_1, j_1 + d_1'r), \alpha_1\beta_1^n\gamma_1), \delta((q, j_1 + d_1r), \alpha_1\beta_1^{n-1}\gamma_1)\}$ contains a final and a nonfinal state. When both d_1r and $d_1'r$ are greater than $|w_1|$, each of these sequences will be either constant or will consist of consecutive integers. If both are constant, w is a witness between any pair of states $(p_1, i_1 + d_1r)$, $(q_1, j_1 + d_1's)$ (since it was a witness between (p_1, i_1) and (q_1, j_1). If either or both sequences consist of consecutive integers, a witness between $(p_1, i_1 + d_1r)$ and $(q_1, j_1 + d_1's)$ is $\alpha_1\beta_1^k\gamma_1$, where $k = \min(m_r, n_s)$.

Let $t_1 = \min(t_0, |w_1| + 1)$, let k_1 be the least common multiple of k_0 and the drops d_1, d_1'.

If there are still unaccounted-for nonequivalences among states, repeat the process on state pairs $(p_2, i_2), (q_2, j_2), \ldots$. The process must halt after only a finite number of steps.□